Thinking Recursively

Part III

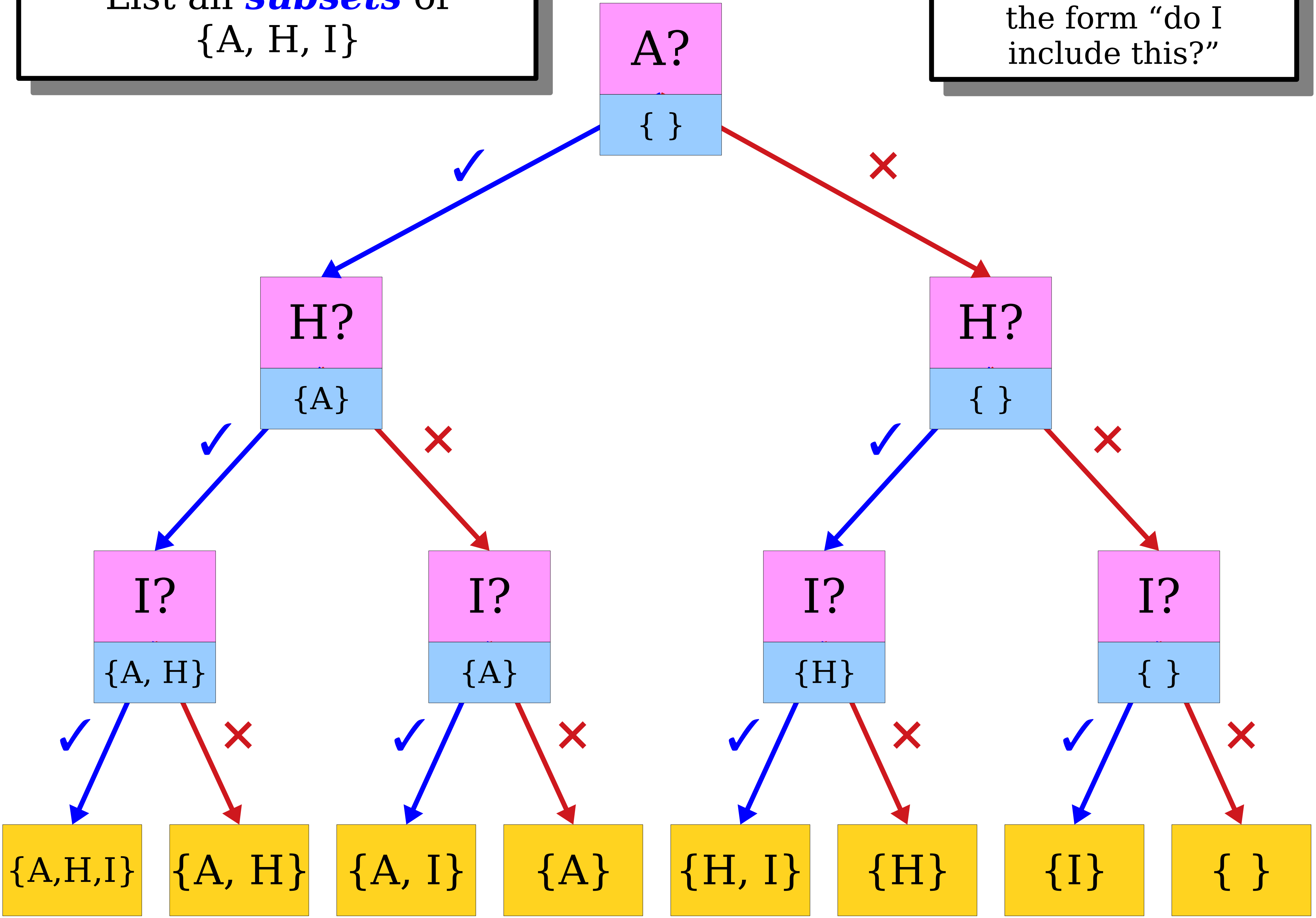
Outline for Today

- ***Recap from Last Time***
  - Where are we, again?
- ***Recursive Optimization***
  - Finding the best solution to a problem.

Recap from Last Time

List all ***subsets*** of
{A, H, I}

Each decision is of
the form “do I
include this?”



A?
{ }

✓ ✗

H?
{A}

H?
{ }

✓ ✗ ✓ ✗

I?
{A, H}

I?
{A}

I?
{H}

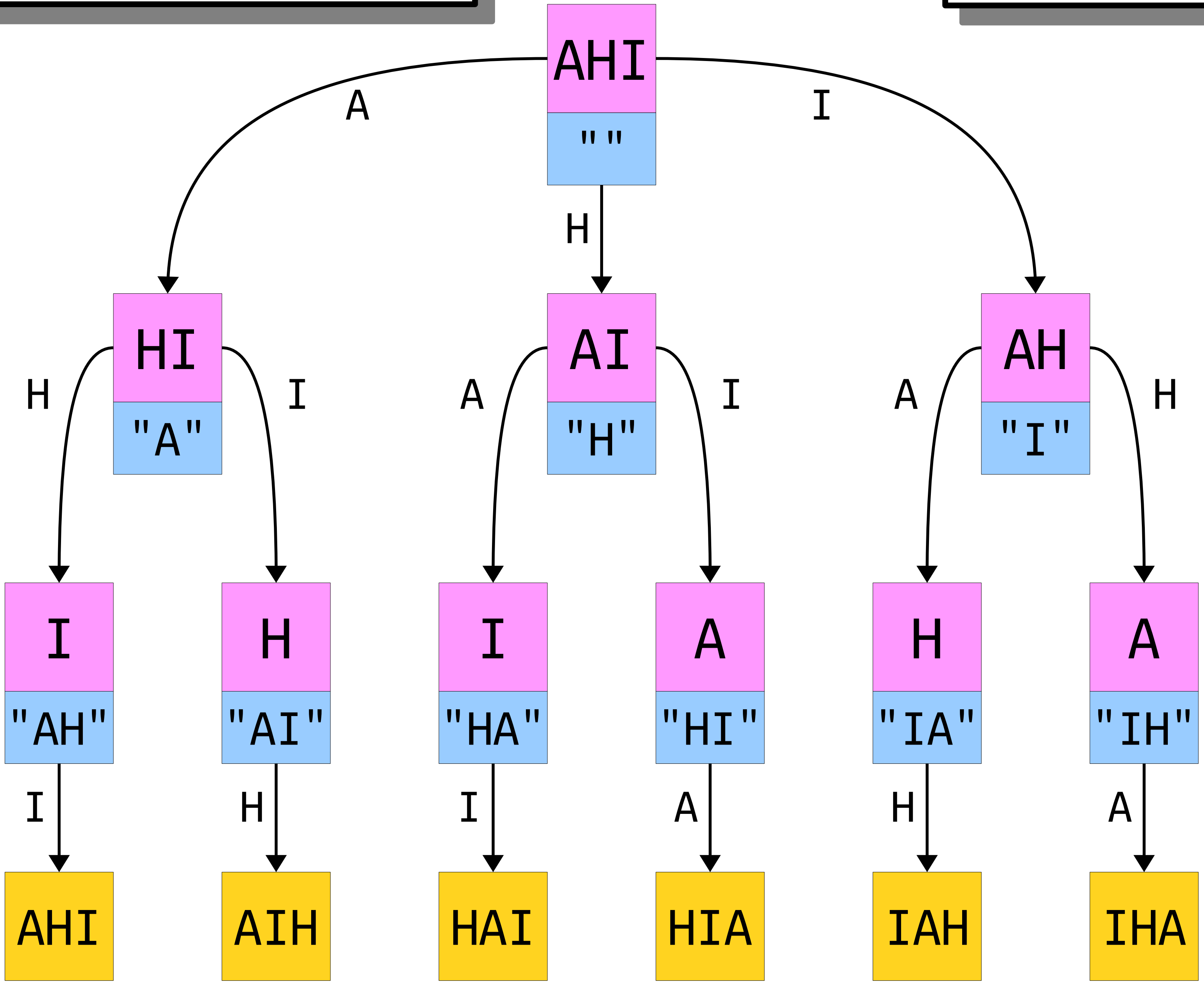
I?
{ }

✓ ✗ ✓ ✗ ✓ ✗ ✓ ✗

{A,H,I}
{A, H}
{A, I}
{A}
{H, I}
{H}
{I}
{ }

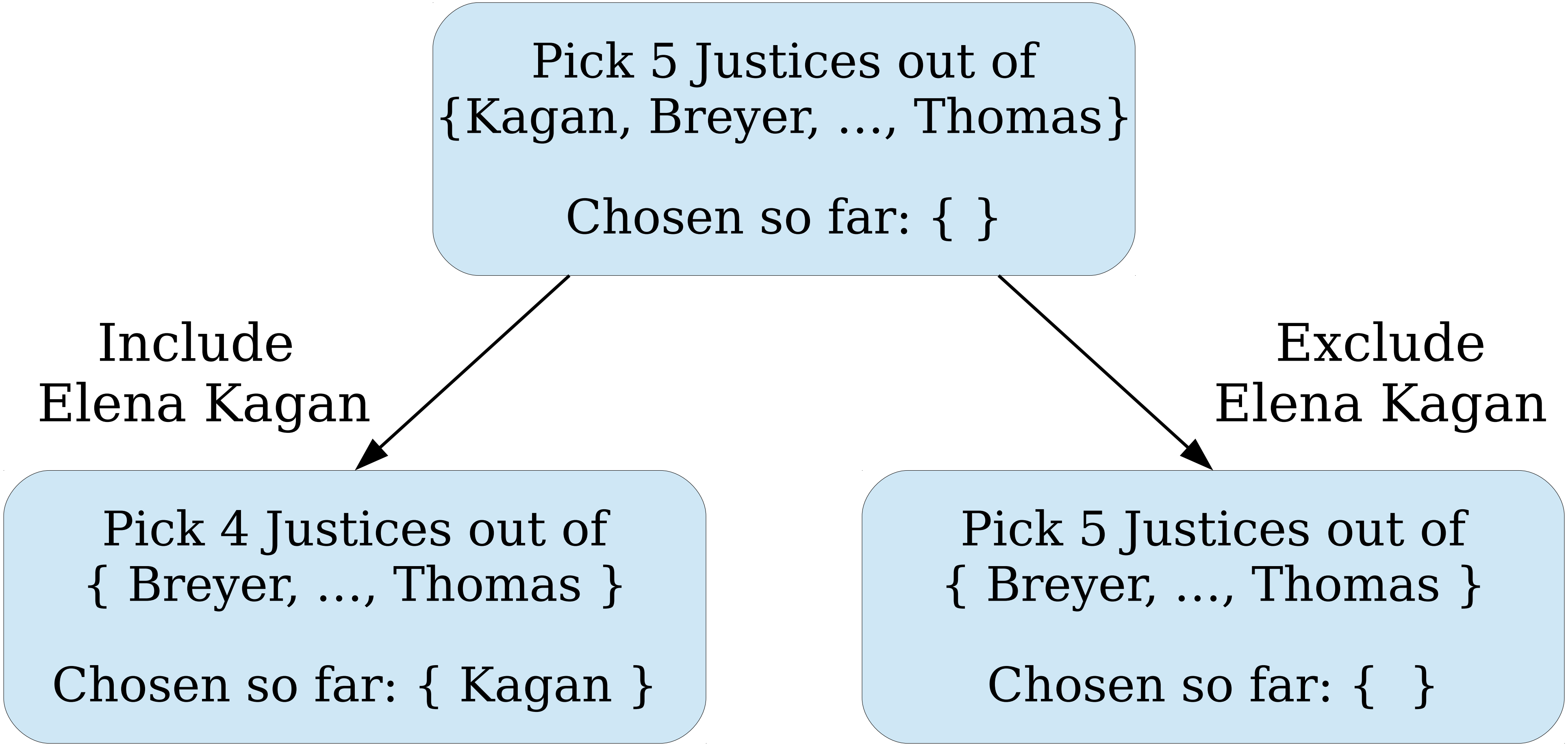
List all ***permutations*** of {A, H, I}

Each decision is of the form “what do I pick next?”



List all ***combinations*** of five justices

Each decision is of the form “do I include this person?”



Pick 5 Justices out of {Kagan, Breyer, …, Thomas}

Chosen so far: { }

Include Elena Kagan

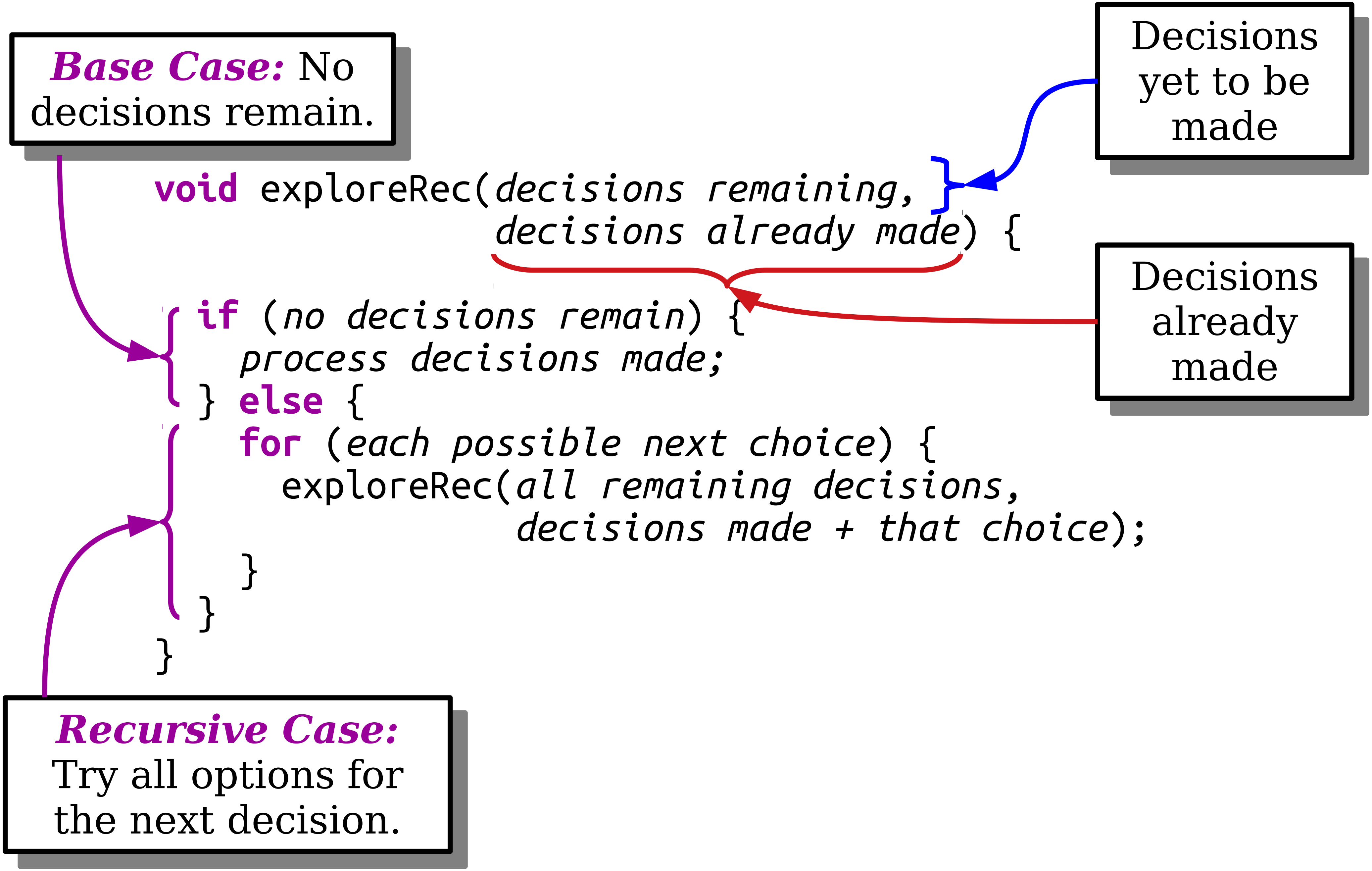
Exclude Elena Kagan

Pick 4 Justices out of { Breyer, …, Thomas }

Chosen so far: { Kagan }

Pick 5 Justices out of { Breyer, …, Thomas }

Chosen so far: { }



Base Case: No decisions remain.

Decisions yet to be made

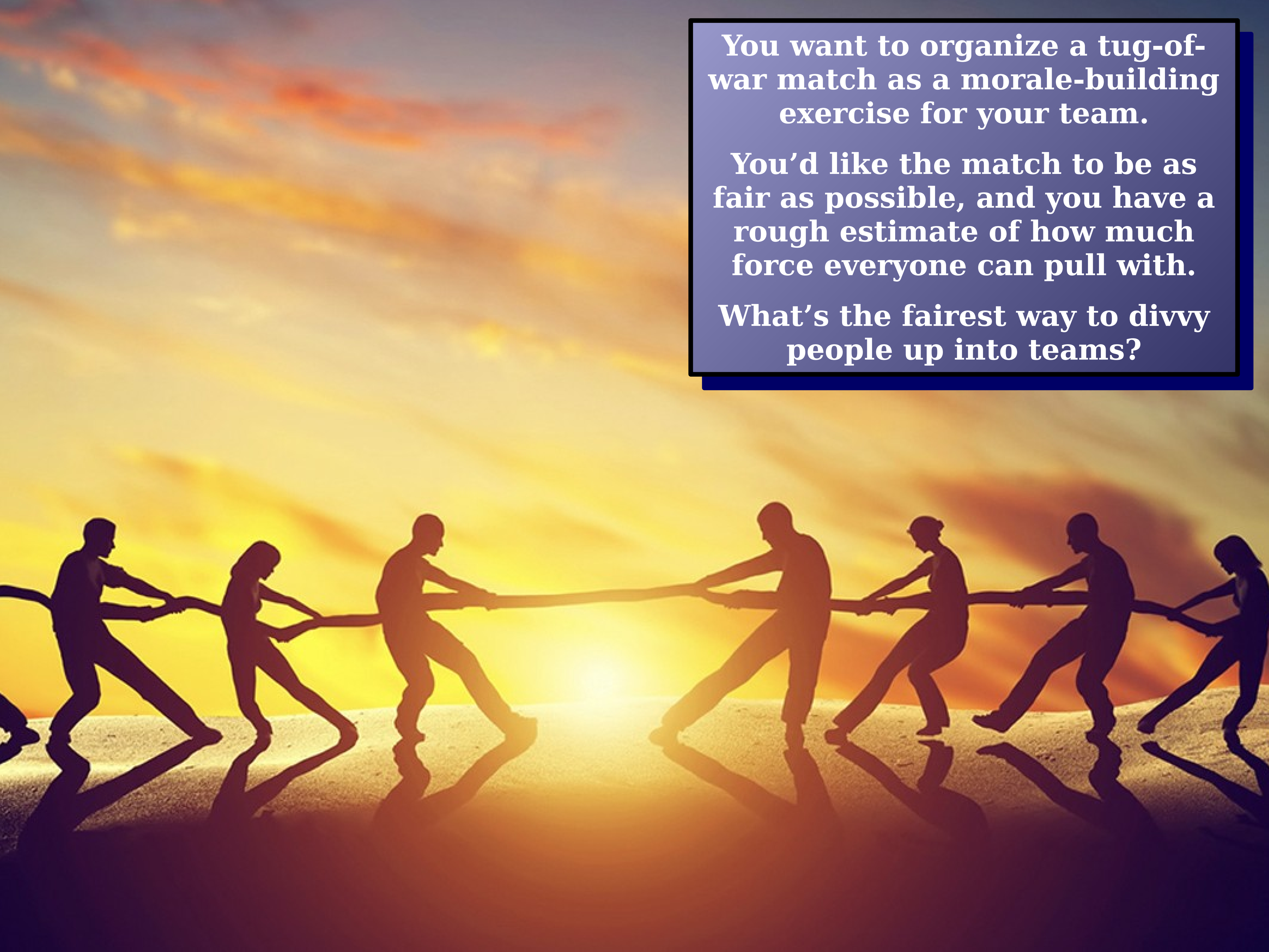
Decisions already made

```
void exploreRec(decisions remaining,
                decisions already made) {
  if (no decisions remain) {
    process decisions made;
  } else {
    for (each possible next choice) {
      exploreRec(all remaining decisions,
                 decisions made + that choice);
    }
  }
}
```

Recursive Case: Try all options for the next decision.

```
void exploreAllTheThings(initial state) {
  exploreRec(initial state, no decisions made);
}
```

New Stuff!

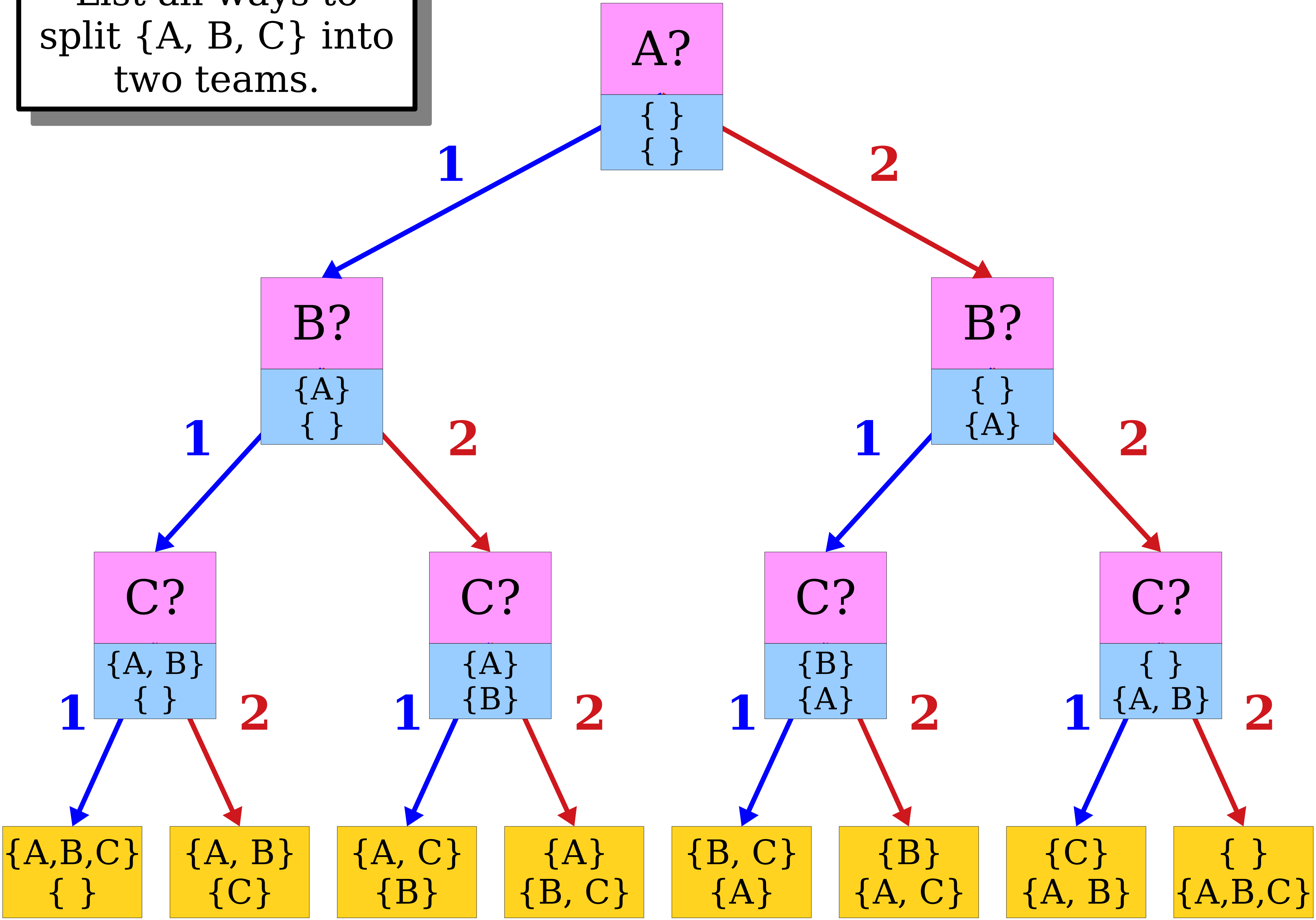


You want to organize a tug-of-war match as a morale-building exercise for your team.

You'd like the match to be as fair as possible, and you have a rough estimate of how much force everyone can pull with.

What's the fairest way to divvy people up into teams?

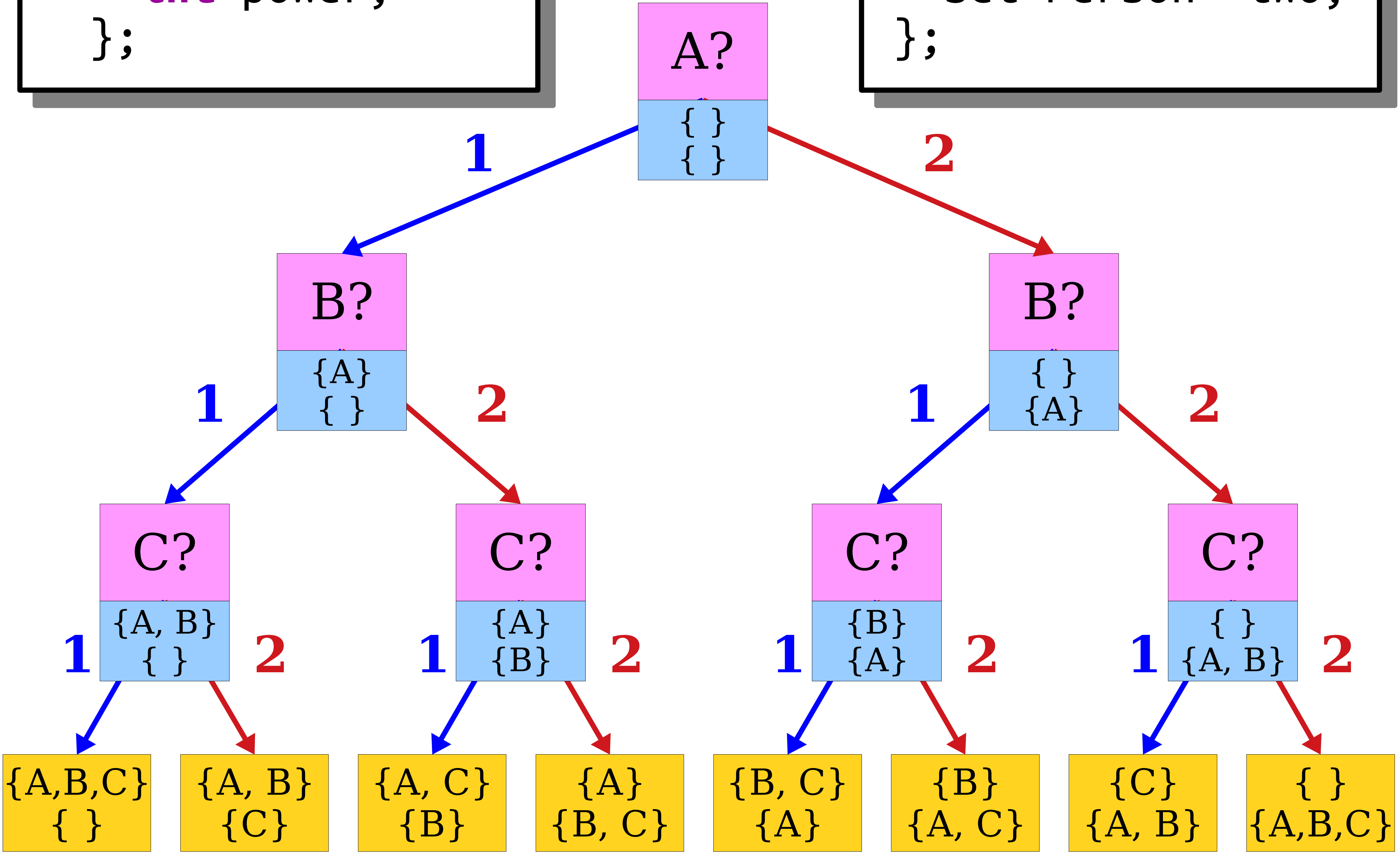
List all ways to split {A, B, C} into two teams.



Let’s Code it Up!

```
struct Person {
  string name;
  int power;
};
```

```
struct Teams {
  Set<Person> one;
  Set<Person> two;
};
```



Tug-of-War

- We currently can list off (***enumerate***) all the ways to split people into two teams.
- At the end of the day, we're only interested in the *most fair* split, not *all possible* splits.
- How can we determine what that split is?

Time-Out for Announcements!

CURIS Applications Open

- CURIS (the Undergraduate Research Internship in Computer Science) is now accepting applications for summer research positions.
- Yes, you can do this with just CS106B!
- For more information, visit

https://curis.stanford.edu

Assignment 2

- Assignment 2 was due at the start of class today.
  - Need more time? One late day will extend the deadline to Wednesday, and a second will extend it to Friday.
- Feel free to use late days without giving us a heads-up over email. We'll do all the appropriate recordkeeping.

Assignment 3

- Assignment 3 (***Recursion!***) goes out today. It’s due Wednesday, February 6, at the start of class.
- Play around with recursive problem-solving across four problems:
  - ***Siepinski Triangle:*** A famous self-similar fractal.
  - ***Human Pyramids:*** Gymnastics meets computer science.
  - ***Shift Scheduling:*** How to maximize profits, and why you might not want to.
  - ***Riding Circuit:*** Justice delayed is justice denied.
- You are allowed to work with a partner on this assignment, though it’s not required. Feel free to use Piazza to find someone to work with!

YEAH Hours

- We will be holding YEAH Hours (Your Early Assignment Help Hours) for Assignment 3. They'll be held

  Tuesday, January 29th
  at ***7:00PM***, in
  room ***380-380X***.

- Can't make it? No worries! Slides will be posted on the course website.

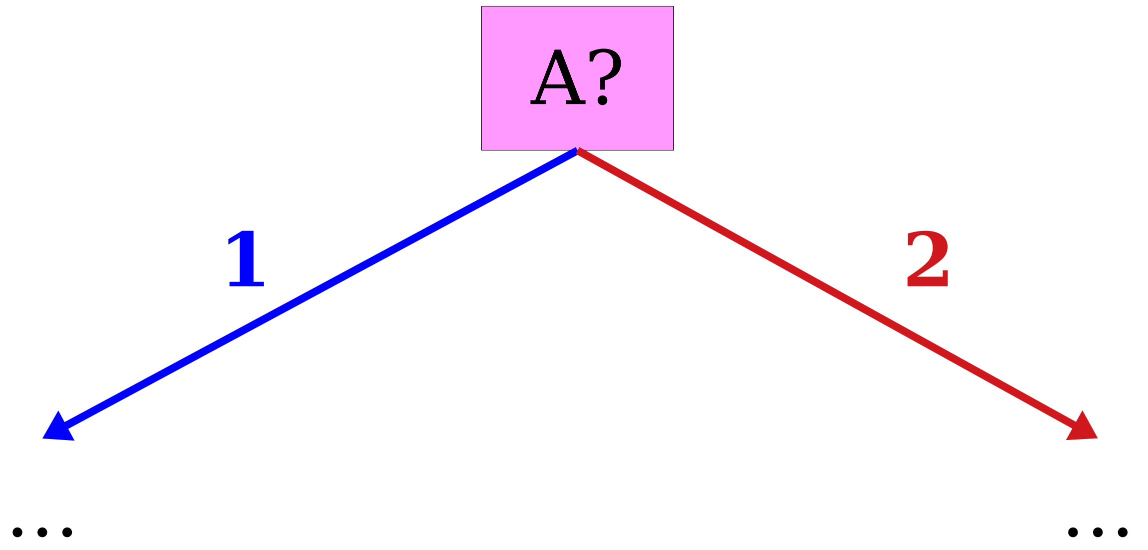
`fg`

(“Foreground;” The UNIX command to resume a program that’s been paused.)

Recursive Optimization

Enumeration and Optimization

- An ***enumeration*** problem is one where the goal is to list all objects of some type.
- An ***optimization*** problem is one where the goal is to find the best object of some type.
- We’ve seen many examples of enumeration problems. How do we solve optimization problems?

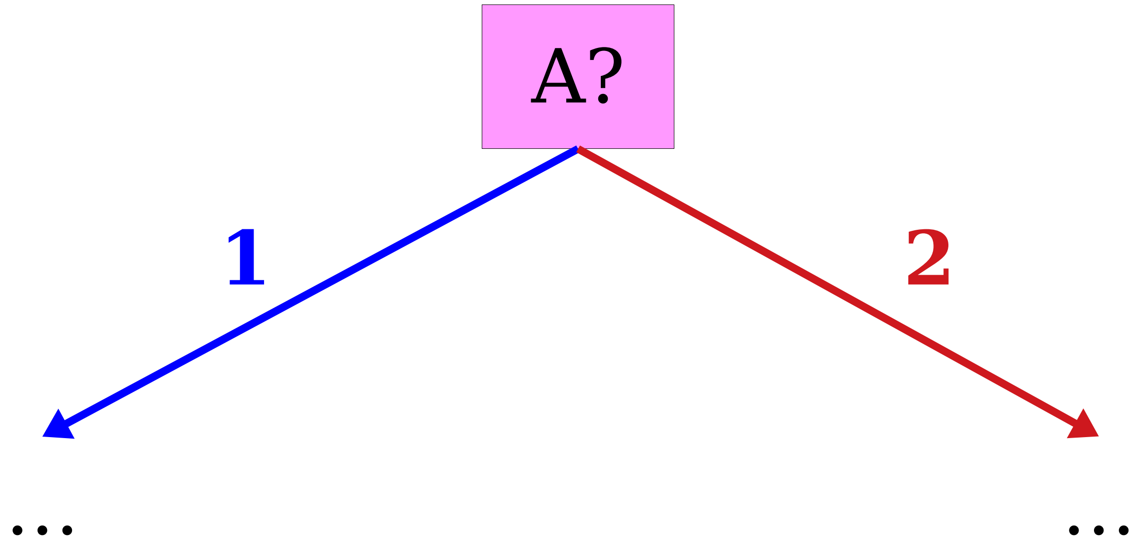


Person A either gets assigned to Team 1 or gets assigned to Team 2.

Therefore, to list all possible splits, we can

- list all splits where A goes on Team 1, then
- list all splits where A goes on Team 2.

Since this covers all possible options, this lists all possible splits.



The best split either assigns A to Team 1 or assigns B to Team 2.

Therefore, to find the best possible split, we can

- find the best split where A is on Team 1,
- find the best split where A is on Team 2, then

choose whichever of these two splits is best, since the best option has to be one of those two.

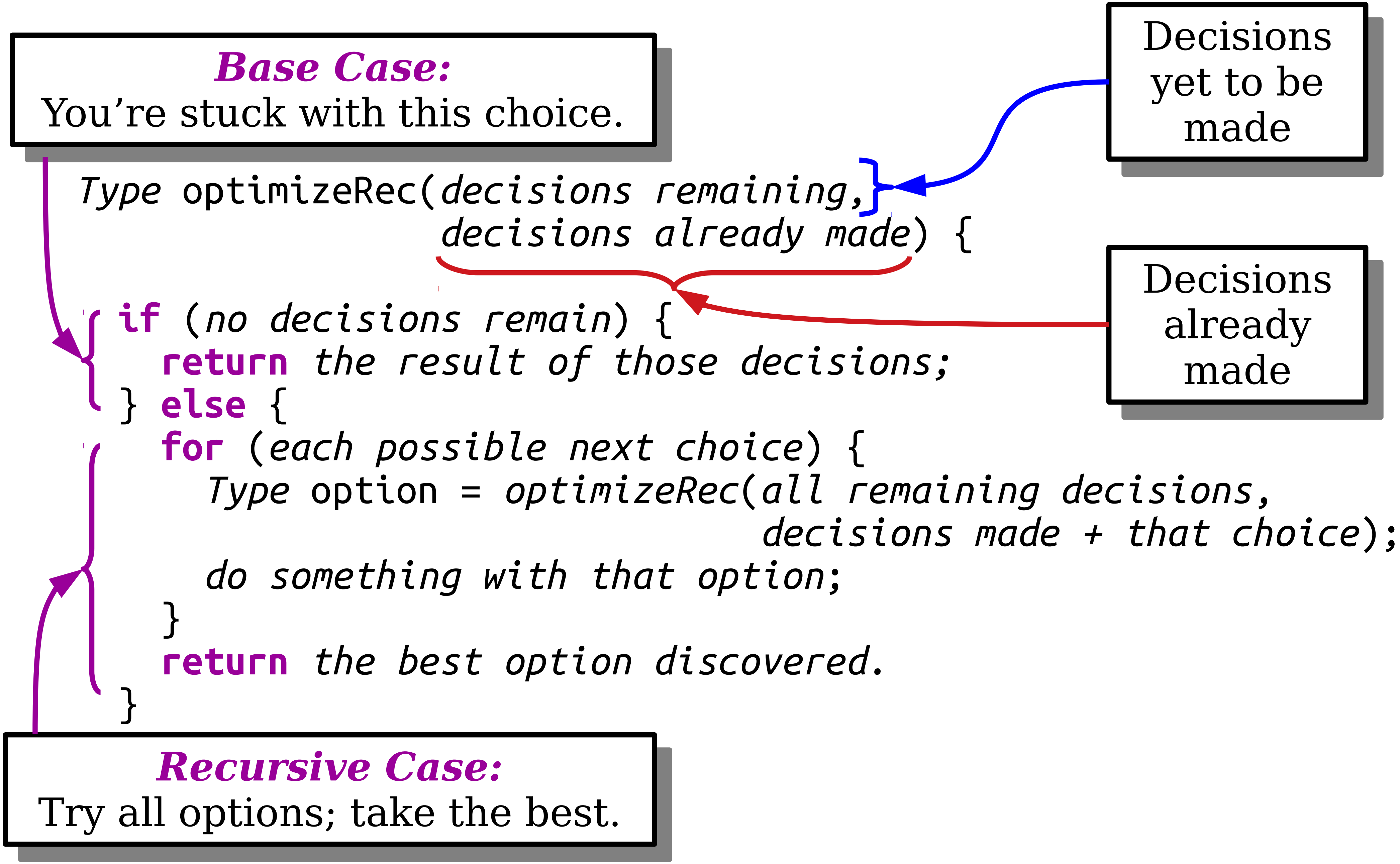
```
Teams bestTeamsRec(const Set<Person>& remaining,
                   const Teams& soFar) {
  if (remaining.isEmpty()) {
    return soFar;
  } else {
    Person curr = remaining.first();

    /* Option 1: Put this person on Team 1. */
    Teams best1 = bestTeamsRec(remaining - curr,
                               { soFar.one + curr, soFar.two });

    /* Option 2: Put this person on Team 2. */
    Teams best2 = bestTeamsRec(remaining - curr,
                               { soFar.one, soFar.two + curr });

    if (imbalanceOf(best1) < imbalanceOf(best2)) {
      return best1;
    } else {
      return best2;
    }
  }
}
```

```
Teams bestTeamsRec(const Set<Person>& remaining,
                   const Teams& soFar) {
  if (remaining.isEmpty()) {
    return soFar;
  } else {
    Person curr = remaining.first();

    /* Option 1: Put this person on Team 1. */
    Teams best1 = bestTeamsRec(remaining - curr,
                               { soFar.one + curr, soFar.two });

    /* Option 2: Put this person on Team 2. */
    Teams best2 = bestTeamsRec(remaining - curr,
                               { soFar.one, soFar.two + curr });

    if (imbalanceOf(best1) < imbalanceOf(best2)) {
      return best1;
    } else {
      return best2;
    }
  }
}
```

This is basically the same code as before! The only difference is that we propagate values back up the recursion.

Recursive Optimization

- The code we’ve written here is an example of a ***recursive optimization***.
- The major change is how the recursive step works.
  - In recursive *enumeration*, the recursive step tries all options for the current decision.
  - In recursive *optimization*, the recursive step does this, but then returns the best solution out of the options it found.



Base Case:
You're stuck with this choice.

Decisions yet to be made

Decisions already made

```
Type optimizeRec(decisions remaining,
                 decisions already made) {
  if (no decisions remain) {
    return the result of those decisions;
  } else {
    for (each possible next choice) {
      Type option = optimizeRec(all remaining decisions,
                                decisions made + that choice);
      do something with that option;
    }
    return the best option discovered.
  }
```

Recursive Case:
Try all options; take the best.

```
Type optimizeAllTheThings(initial state) {
  return optimizeRec(initial state, no decisions made);
}
```

Your Action Items

- ***Start working on Assignment 3***
  - Aim to complete the Sierpinski triangle and to have started Human Pyramids by Wednesday.
- ***Review the Cell Towers example***
  - It’s in the lecture on the Vector type. Based on what we’ve covered, does that example make a bit more sense?
- ***Finish reading Chapter 8***
  - There’s plenty of useful insights and ideas in there!

Next Time

- ***Recursive Backtracking***
  - Searching for a needle in a haystack.
- ***The Great Shrinkable Word Problem***
  - Helping your relatives with recursion.